Optimizing the Integration of Agent-based Cloud Orchestrators and Higher-level Workloads

Merlijn Sebrechts, Gregory Van Seghbroeck, and Filip De Turck

Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium
merlijn.sebrechts@ugent.be

Abstract. The flexibility of cloud computing has put significant strain on operations teams. Manually installing and configuring applications in the cloud simply isn't an option anymore. Configuration management automation solves the issue of getting a single application into a certain state automatically and reliably. However, the issue of automatic dependency management between multiple applications is still an "open, hard problem" according to researchers at Google. Agent-based modeling and orchestration tools like Juju solve the issue of getting from zero to a working set of correctly clustered and connected frameworks. The shortcomings of these state-of-the-art tools are that they don't provide efficient ways to model and orchestrate workloads running on top of these frameworks. This paper presents a number of ways to deploy and orchestrate workloads with Juju, compares their performance and overhead, and suggests how this overhead can be minimized.

Keywords: Cloud Modeling Languages, Service Orchestration, Juju

1 Introduction

There is a big need to make IT operations easier. Take the field of data science for example. There is an ever-growing set of tools and platforms that support data scientists. The prevalence of open-source software in that field has shifted the barrier of entry from licensing costs to operations costs. The tools are available and free to use, but actually running them in production requires a team of system administrators that have expert knowledge on both the tools themselves and IT operations in general. Even industry-standard companies such as Google state that the issue of automatic dependency management between multiple services is still an "open, hard problem"[1].

The devops world has spawned a number of useful tools that help operations teams. Configuration management systems help automate the task of installing, configuring and managing applications. Automating these tasks reduces errors and saves a lot of time when scaling an application. This process, called infrastructure as code, allows businesses to quickly react to changes in usage of their application. These languages are less suited to lower the time to market because each new application requires new management code. Moreover, these

tools don't really abstract away the complexity of operations. This means that operators using these tools now have to be experts in three fields: Configuration management, IT operations and the applications they're maintaining.

Cloud modeling languages aim to reduce complexity and time to market by providing an abstraction layer on top of IT operations. Instead of changing the applications themselves, the operator changes a model that represents the application. The orchestrator then translates actions on the model into actions on the application. This is a great step forward to manage the complexity of IT operations. The current generation of cloud modeling languages such as OASIS TOSCA[2] also improve flexibility and re-usability of operations code by dividing the operations code of an entire cloud application into a number of reusable isolated pieces connected to each other using dependencies.

Monolithic cloud orchestrators have a tendency to become very complex[3]. This results hard-to-maintain and hard-to-scale bottlenecks. Agent-based orchestrators such as Juju [6] are the solution to this problem. All the dependency resolution and operations logic is put into a series of agents that communicate with each other over predefined interfaces. The only responsibility of the orchestrator is to install the agents and set up communication channels between them[5]. The actual dependency resolution happens in the agents. This has the added benefit that the implementation of the agent is hidden. This makes it possible for two agents that manage services using two different configuration management tools to communicate with each other, exchange information, and feed that information into the config management tools.

The combination of agent-based cloud orchestrators and cloud modeling languages makes IT operations a lot easier but there is still a lot of work to be done. All the aforementioned tools have a strong focus on the operations of an application as a combination of services. What is left out are the actual workloads running on top of these services. It's great that orchestrators allow an operator to setup a MySQL database, but what about the tables in the database? It's easy to model and orchestrate an Apache Hadoop cluster, but what about the jobs running on top of that Hadoop cluster? This isn't only about creating the table and submitting the job. The MySQL table will be used by some software or algorithm and the Hadoop job will get data from somewhere and put the extracted information somewhere else. Configuring all these workloads by hand isn't a viable option due to the same reason that running the operations of an entire application isn't a viable option: it's error-prone, it slows innovation down to a crawl, and requires a very competent team with highly specialized skills.

Since agent-based cloud orchestrators solve these challenges for the operation of services and applications, they form a great start to explore solutions for the operation of high-level workloads.

2 Modeling high-level workloads in Juju

The authors' previous work proposed the workflow component as a way to model and manage high-level workloads with Cloud Modeling Languages[4]. Each work-

flow component is a Charm that contains both the workload itself and a workflow agent that manages the workload. This approach provides a lot of flexibility without adding any additional logic to the Juju orchestrator itself. The tricky part of this approach is that each workload requires at least one agent, and this agent needs to run somewhere. Juju provides two ways to run additional agents: co-locate the workflow agent and the framework agent without any isolation and isolate the workflow agent from the framework agent by running it inside an LXD container.

Both methods aren't ideal. It clearly shows that Juju is not built with such use in mind. The issue with co-location is that Juju doesn't allow two co-located agents to run in parallel. This is to avoid conflicts when two agents try to manage the same machine at the same time. This significantly slows down the agents because each agents needs to wait for the other agents to finish executing. Isolating the agents using LXD containers solves this issue but introduces a new one: the overhead of the LXD container. In many cases the overhead of the LXD container is larger than the resources used by the actual workload.

3 LimeDS Big Data model

This paper evaluates both methods for running additional agents in order to get a better grasp on what the actual overhead is and how it compares to the resources used by the workload. The evaluation is done using the LimeDS Big Data model[1]. This model and its components is further explained in this section.

LimeDS is a modular platform to create and run data-driven services[2]. The LimeDS Big Data model is perfect for validating the flexibility of modeling workloads for a number of reasons. First of all, **LimeDS is both a workload and a platform**. The LimeDS Docker container is a workload running on top of the Docker host, but it is also a host to services and modules running on top of LimeDS. It is important to support such flexibility. Having LimeDS and the Docker runtime be two different Charms also has the advantage that you can swap out the single Docker host and plug in for example a Kubernetes cluster. Secondly, **workloads running on LimeDS need to connect to other services**, for example external datastores or load balancers. These connections require a workload agent communicating with other services to exchange the correct information and to resolve possible dependencies such as the workload having to wait for MongoDB to start. Lastly, **LimeDS needs to run in a scaled-out setup** to handle Big Data workloads. The agents make this incredibly easy. An operator specifies how many instances of LimeDS are needed. The Orchestrator installs an agent for each LimeDS instance, and the agents communicate with the Docker host agent to deploy LimeDS correctly. Since each LimeDS agent implements the http interface, the agents don't need any addtional clustering logic. Each agent connects to the agent managing the HAProxy load

[1] https://jujucharms.com/u/tengu-team/limeds-bigdata/
[2] http://limeds.be/

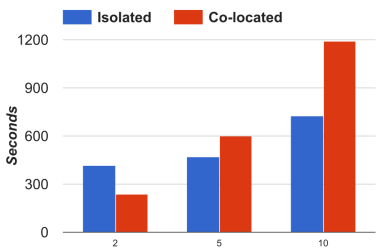

Fig. 1. The deploy-time overhead of the agents; LXD vs co-located.

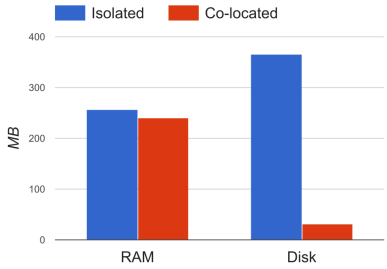

Fig. 2. The runtime overhead of the agents; LXD vs co-located.

balancer, and that agent configures the proxy correctly to loadbalance requests over the LimeDS cluster.

4 Evaluation

The **deploy-time overhead** is measured as the time it takes from the model to scale. The tests start with a running LimeDS Big Data cluster with two units. This cluster is then scaled to n units, and the time until the scaling action is complete is measured and compared.

The results in Figure 1 show that the deploy-time overhead is initially greater for the isolated setup than for the co-located setup. This is due to the overhead of spinning up an LXD container for the new agent. However, when more units are requested, the isolated setup scales faster than the co-located setup due to the sequential nature the co-located setup. Only one co-located agent is allowed to execute actions at any given moment.

For the **runtime overhead** of the agents, the memory and disk usage of the agent are recorded as shown in Figure 2. Here the disadvantage of the isolated setup is clearly visible, it has a much bigger runtime overhead. The +200MB of RAM usage per agent is especially worrisome since the LimeDS container itself uses about 300MB of RAM.

5 Conclusion and the road forward

Neither of the solutions has satisfactory performance. The co-located setup compromises heavily on deploy-time overhead and both setups compromise on runtime overhead. There are a few advantages to these solutions. Having the ability to write arbitrary logic in the agent enables complex dependency resolution without adding complexity to the orchestrator itself. Containers successfully stop the workload agents from accessing or changing the machine where the framework is running. This forces agents to communicate using the relationships. This enables other frameworks to implement the same relationship, making the solution

pluggable. The ability to model higher-level workloads as a combination of components related to each other gives operators a clear view of what is actually running, and allows the workloads themselves to be pluggable. The challenge will be to find a solution that addresses the performance issues described here without compromising on the stated advantages. Future research will explore the road forward in a few directions.

Agentless Agents: The advantage of the agents is that they allow running arbitrary dependency handling code, thus keeping the orchestrator simple. A possible solution might be to have a way for giving snippets of dependency handling code to other agents instead of spinning up new agents.

Slim Agents: Instead of reducing the amount of agents, another path forward is to investigate if the overhead of the agent itself can be reduced. This approach requires thorough investigation into where the overhead comes from. There is also potential to use more lightweight process containers such as Docker instead of the full-blown operating system containers that LXD provides.

Parallel co-located Agents: Enabling co-located containers to run in parallel is a possible solution to the deploy-time overhead of co-located agents. This would need each agent to specify what kind of operations the agent will execute. The orchestrator can then use that information to determine whether or not two agents are allowed to run at the same time.

Acknowledgment Part of this work has been funded by the iFest project, co-funded by imec and VLAIO.

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Queue 14(1), 70–93 (Jan 2016), http://dl.acm.org/citation.cfm?doid=2898442.2898444
2. OASIS: TOSCA Simple Profile in YAML Version 1.0 (Aug 2016), https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html
3. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 351–364. EuroSys '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2465351.2465386
4. Sebrechts, M., Borny, S., Vanhove, T., Seghbroeck, G.V., Wauters, T., Volckaert, B., Turck, F.D.: Model-driven deployment and management of workflows on analytics frameworks. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 2819–2826 (Dec 2016)
5. Sebrechts, M., Vanhove, T., Van Seghbroeck, G., Wauters, T., Volckaert, B., De Turck, F.: Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration. In: proceedings of the 2016 IEEE International Conference on Mobile Services (MS 2016). IEEE (2016)
6. Tsakalozos, K., Johns, C., Monroe, K., VanderGiessen, P., Mcleod, A., Rosales, A.: Open big data infrastructures to everyone. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 2127–2129 (Dec 2016)